DRAFT : SYMBOLIC OPTIMIZATION AND ANALYSIS OF NUMERICAL DYNAMICS SOLUTIONS

James Critchley
Multibody.org
Lake Orion, Michigan, USA

Amy Critchley
Multibody.org
Lake Orion, Michigan, USA

Paramsothy Jayakumar
US Army TARDEC
Warren, Michigan, USA

ABSTRACT

A symbolic interpreter for the optimization and analysis of numerical dynamics solutions is presented. The interpreter is intended to address the order of magnitude performance disconnect between the numerical environments which support most general purpose dynamics implementations and the speed of heavily customized domain specific tools and/or symbolic implementations. Unlike other symbolic environments, the objective of this implementation is to be integrated and interchangeable within procedural numerical source code thereby enabling an auto-code feature within any software with only minor cosmetic modifications. Specifically, the interpreter generates efficient simulations via symbolic optimizations which include removal of trivial and identity operations, removal of redundant calculations, identification of constants and preprocessing steps, and culling of extraneous computations. The construction of the directed graph of symbolic computations also presents interesting opportunities for complexity analysis, optimal hybrid methods, and tailored fine grain parallel implementations.

INTRODUCTION

The benefits of symbolic optimization applied to multibody equations of motion are well documented in the literature. Many symbolic solutions have demonstrated in excess of an order of magnitude speed increase (SD/Fast[1], AUTOLEV[2], and DynaFlex/Pro[3], to name but three) while also presenting unique advantages when analysis requires more information than advancing the system state (such as design sensitivity information and nonlinear filtering).

Symbolic optimization reduces the run-time of any numerical procedure by identifying and removing trivial operations and redundant calculations, identifying and isolating values which are constant and/or may be preprocessed, culling extraneous computations which do not participate in the solution, and replacing the general computation pipeline, which is full of conditionals and indexing, with the exact sequence of operations. The relatively small size of multibody problems and the large potential for optimizations renders them ideal candidates for symbolic implementation.

The lack of adoption of symbolic methods can be attributed to several factors. One important point is that symbolic solutions cannot gracefully handle changes in system topology. Instead, a symbolic solution must reinterpret the system, reformulate, optimize, and recompile the solution. As a preprocessing step, the user wait time for this process is commonly acceptable. However, it is not feasible to include such a process within the execution of a dynamic simulation.

Another important factor is the inclusion of forcing functions. Many force elements such as dynamic contact do not result in clean symbolic implementations. The solution is to use a numerical library for the contact force calculations.
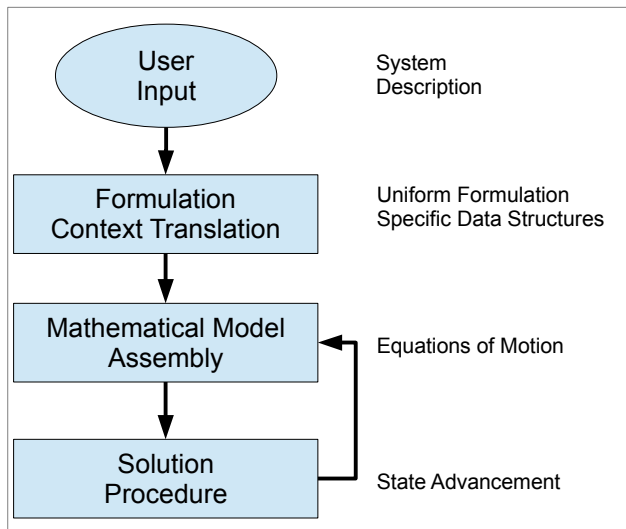
And lastly, one must consider the numerical nature of the final solution itself. A fully automated symbolic multibody solution cannot apply the most robust solution methods within the symbolic context. The equivalent of a generalized mass matrix inversion must proceed numerically to avoid ill-conditioning. For example, the partitioning of coordinates into independent and dependent can result in formulation singularities where the simulation becomes invalid near certain operating points. A fully symbolic implementation then cannot be as robust as a numerical solution and hybrid solutions which rely on numerical solution (inversion equivalent) schemes are employed in practice.

These issues with symbolic multibody methods are perceived as limiting application to the most general simulation cases, but should also be understood as exceptional cases. And

Report Documentation Page

1. REPORT DATE **31 JAN 2013**	2. REPORT TYPE **Journal Article**	3. DATES COVERED **15-08-2012 to 29-11-2012**

4. TITLE AND SUBTITLE **SYMBOLIC OPTIMIZATION AND ANALYSIS OF NUMERICAL DYNAMICS SOLUTIONS**	5a. CONTRACT NUMBER **W56HZV-12-C-0219**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S) **James Critchley; Amy Critchley; Paramsothy Jayakumar**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC,6501 East Eleven Mile Rd,Warren,Mi,48397-5000**	8. PERFORMING ORGANIZATION REPORT NUMBER **; #23637**

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC, 6501 East Eleven Mile Rd, Warren, Mi, 48397-5000**	10. SPONSOR/MONITOR'S ACRONYM(S) **TARDEC**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **#23637**

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES **For Proceedings of the ASME 2013 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2013**

14. ABSTRACT

A symbolic interpreter for the optimization and analysis of numerical dynamics solutions is presented. The interpreter is intended to address the order of magnitude performance disconnect between the numerical environments which support most general purpose dynamics implementations and the speed of heavily customized domain specific tools and/or symbolic implementations. Unlike other symbolic environments, the objective of this implementation is to be integrated and interchangeable within procedural numerical source code thereby enabling an auto-code feature within any software with only minor cosmetic modifications. Specifically, the interpreter generates efficient simulations via symbolic optimizations which include removal of trivial and identity operations, removal of redundant calculations, identification of constants and preprocessing steps, and culling of extraneous computations. The construction of the directed graph of symbolic computations also presents interesting opportunities for complexity analysis, optimal hybrid methods, and tailored fine grain parallel implementations.

15. SUBJECT TERMS					

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT **Public Release**	18. NUMBER OF PAGES **7**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

in the remainder of this paper, the emphasis will be on the similarities of symbolic and numerical multibody solutions.

In what follows, the design considerations of a software system for the symbolic interpretation of numerical multibody methods will be discussed. The implementation will then be summarized and followed by a complete illustrative example using direction cosine matrices. The results of an application to a scalable multibody problem are then presented, followed by outlines for future work and conclusions.

DESIGN

Multibody simulators with general system applicability follow a regular pattern. The description of the system to simulate is provided as input and system element (body, joint, force, etc.) are then converted into uniform formulation specific data structures. The formulation is then applied to the data to form a mathematical model, which is subsequently solved. It often happens that these last these last two steps actually run concurrently.
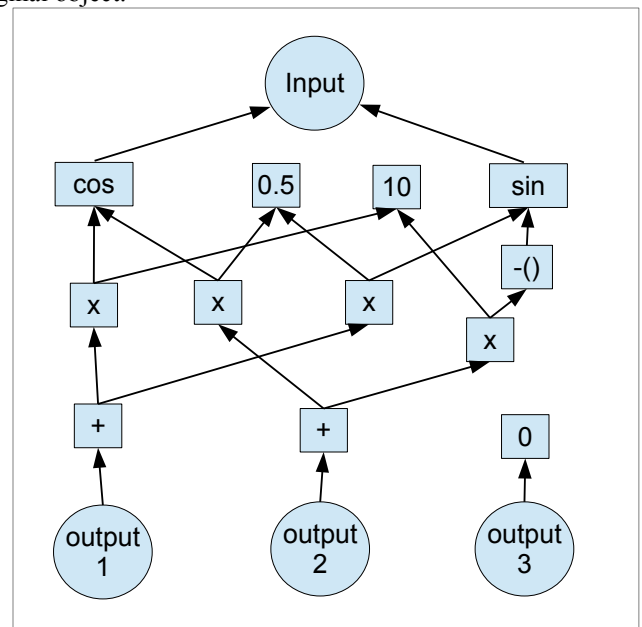

Figure 1: A pattern for generic multibody solutions.

In a numerical implementation, the logic associated with the system assembly operation is repeated continuously as the system states are updated. The only difference in a symbolic implementation is that the problem description is assumed to be static and only the floating point operations with direct dependence on the evolving states are repeated after being initially interpreted.

Recognizing that the multibody problem formulation for symbolic and numerical methods is largely identical, a simple method for generating both from the same software implementation is sought. Specifically, adding the performance benefits of symbolic optimization to a numerical software package will be explored. The other two integration possibilities; adding general numerical procedure to symbolic software; or creating an intermediate representation able to generate both, will not be discussed.

In symbolic mode, the software shall be configured to interpret and record all floating point operations as symbolic

objects. With the interpreter active, the multibody system will be setup with additional input from the analyst identifying the various inputs as constants, parametric constants (which may change from run to run, but not during a simulation), or variables (such as externally applied forces). The multibody system is then solved (or mass matrix formed, etc.) and the desired solution elements identified as output. Having captured the symbolic form of numerical implementation, the systems will then optimize it, and write out source code for compilation. The recorder feature will operate only on the floating point numbers and the logic which organizes these numbers is assumed to be static.

From a procedural perspective, the system and solution shall be configured and execute the same command sequence in both symbolic and numerical modes. The validity of the symbolic process can then be checked at by running the numerical algorithm in place of the symbolic solution.

The type of symbolic optimizations which are required for this task do not require a state of the art computer algebra system. The symbolic capabilities which are sought are:

1. removal of trivial operations
2. removal of redundant calculations
3. identification of constants
4. identification quantities which may be preprocessed
5. culling of extraneous computations
6. fully numerical pipeline (no logical decisions).

Sophisticated symbolic manipulation tasks such as expression simplification and differentiation are notably absent, keeping this a simple and lightweight application.

A basic symbolic processor performs these functions by loading constants, variables, and their relationships into memory and avoiding duplication via hash consing[4]. For example the constant zero will be created only once, and subsequent attempts to create a zero constant will generate the same hash key and instead return with a reference to the original object.

Figure 2: A simple dependency graph.

Trivial and identity operations are removed as manipulations are loaded into memory. For example a request to multiply by 1.0 will not load anything into memory and instead return the original object.

Applying attributes such as "constant" and "variable" to input expressions allows such attributes to be propagated automatically to all subsequent calculations using simple rules. For example an expression containing only constant inputs is a constant, and any expression containing a variable acquires the variable attribute.

To capture quantities which may be preprocessed, a third attribute called "parametric constant" is introduced which denotes values which are constant during simulation and can be preprocessed.

As a solution is loaded into memory, it forms a dependency graph where every operation is stored with reference to (linked to) its input. A sample dependency graph is visualized in figure 2. In this way, the final step after interpreting a solution procedure is to identify the values which will be output. In a dynamic simulation outputs may be the system state, complete kinematic information of the body reference frames, or anything of interest.

Beginning with the output, the dependency graph can be traced backwards to terminal inputs and constants. This procedure identifies only the operations which directly participate in the calculation of the output and permits large sections of the stored graph to be culled (ignored). In the case of the dependency graph shown in figure 2, many computations would be ignored if "output 2" were no longer of interest. From this perspective a general numerical multibody solution that calculates complete kinematic information uniformly for all declared points will, in symbolic form, only compute the positions, velocities, and/or accelerations as necessary to satisfy the requested outputs.

In a final step, the culled list of symbolic expressions will be used to construct program source code which can itself be compiled and executed. This is to be accomplished by reusing a simple mainline program template and printing out each element's definintion as it appears in the list.

Output for C language is the initial objective, however export to other languages such as FORTRAN, JAVA, Python, m-script, or to more capable symbolic processors such as Maple or Mathematica follows analogously.

IMPLEMENTATION

The target numerical multibody solution to which the symbolic optimization will be added is implemented in the C++ programming language. C++ allows for overloading of all common arithmetic operations and in this way, an object can completely replace the standard *double* or *float* type identifiers. The object which assumes this responsibility will be the class *RealNumber*.

Rather than parse the source code with a global find and replace action, it is common for software engineers to use compile time switches and type definitions to easily adapt software to different target platforms. In this case the defined type replacing all floating point values is *Real* and is usually implemented for double precision computing as

```
typedef double REAL;
```

but will instead be implemented with conditional behavior.

```
#ifndef MBD_SYMBOLIC
  typedef double REAL;
#else
  typedef RealNumber REAL;
#endif
```

The symbolic condition can be activated at compile time using the switch *-dMBD_SYMBOLIC*. This allows the solution library to be compiled into two distinct modes, numerical and symbolic which can exist simultaneously.

With the availability of single switch substitution of a fundamental symbolic object for all floating point types, the *RealNumber* class must then provide access to symbolic capabilities. To do this a single level of indirection is required to overcome a few obstacles in the C++ language semantics. It is expected that the target software manipulate all symbolic quantities uniformly and generate different results depending on the type. This is a textbook application of polymorphism where in C++ a base class defines an interface and the variants inherit and define the behaviors which make them unique [4].

Unfortunately the language makes straightforward application difficult because as a *RealNumber* takes the place of a traditional floating point, it will commonly be initialized then set to a value later, or even continually reused. In the context of the symbolic solution this would require that the object be created with one symbolic type then later updated to something else. In C++, a variable cannot change representations and the easy way around this is to use a generic pointer. However, C++ operator overloading does not apply to pointers. Using a reference instead would satisfy the requirements for arithmetic operator overloading but constructors and assignments would not be straightforward (if not syntactically impossible).

Hence the RealNumber class serves as container for a symbolic expression object (*SymbolicExpr)* and provides its interface to operator and function overloading. The *RealNumber* class is shown in the collaboration diagram of figure 3.
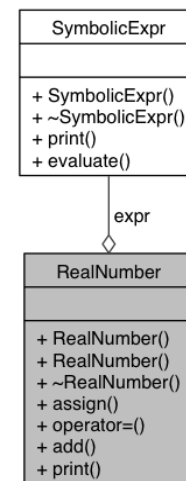

Figure 3. Collaboration diagram for *RealNumber*.

The *SymbolicExpr* class is the fundamental abstract symbolic unit. It is an abstract base class, meaning it cannot be instantiated, and instead completely defines the the interface which all concrete symbolic expressions must share. For example, a *SymbolicConstant* is one of many a concrete symbolic expression objects. The *SymbolicConstant* inherits its interface from (is derived from) *SymbolicExpr* and provides a real implementation which can be executed. The inheritance is depicted in the *SymbolicExpr* class diagram (which is abbreviated for presentation).

Figure 4: Class diagram for *SymbolicExpr*.

A *SymbolicExpr* responds to high level inquiries about its attributes, with methods such as *is_negative*, *is_constant*, *is_parametric*, and *is_variable*. These attributes define common types, but are also applied to the results of arithmetic and function calls. As mentioned earlier, the attributes allow decisions such as the addition of two *SymbolicConstant* objects to produce a new *SymbolicConstant* while the addition of a *SymbolicConstant* and a *SymbolicVariable*, produces a *SymbolicAdd* object with an attribute indicating it is variable.

A symbolic expression must also provide a *name* and be able to *print* itself in C syntax.

Currently the *SymbolicExpr* hierarchy is flat. However, there is significant commonality across functions (such as *sin*, *cos*, *abs*, and *exp*) and the arithmetic operators. Additional abstract classes derived from *SymbolicExpr* are under consideration. Aside from providing a convenient grouping of common functionality when implementing a new symbolic function, such a change would not alter the architecture.

Symbolic expression objects are collected and managed by a single object, called a Symbolic Expression Pool (*SymbolicExprPool*) which ensures that no duplicate symbolic expressions are created. The pool is a hash based index of every computation which has been loaded into memory. Provided each computation has a unique and repeatable hash, a pre-existing object can found and returned.

The *SymbolicExprPool* is implemented as a singleton class which allows only a single instance of the pool to be used at run time [4]. In other words an attempt to create a new *SymbolicExprPool* object will return with the same *SymbolicExprPool* object which is currently in use in other parts of the program (e.g. there is only one global *SymbolicExprPool*). For this reason access to the pool is guarded by a mutual exclusion, *mutex,* which prevents simultaneous reads and writes resulting in collisions or inconsistent data and behavior[5].
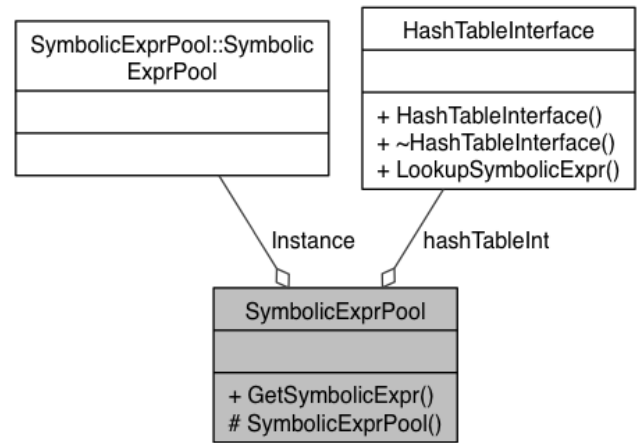

Figure 5: Collaboration diagram for *SymbolicExprPool*.

The *SymbolicExprPool* uses a generic hash table interface (*HashTableInterface*) which is currently implemented using the Standard Template Library map (*std::map*)[6].
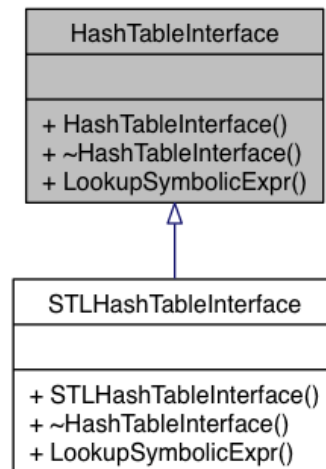

Figure 6: Class diagram for *HashTableInterface*.

The *SymbolicExprPool* provides a method, *get_symbolic_expr()* which uses a key string to search for the matching *SymbolicExpr* in the pool. If one such match exists, the *get_symbolic_expr()* method returns the found *SymbolicExpr* and produces *NULL* otherwise. This model allows the directed graph of simulation computational dependencies to be constructed without duplication.

A SIMPLE EXAMPLE

The process flow and basic use of the symbolic capability will be demonstrated with a simple matrix example. Three by three matrices are commonly used in numerical dynamic simulations to represent quantities such as basis transformation, the skew symmetric cross product operation, and the inertia tensor. It is common for such matrices to contain zeros, ones, symmetry, or other duplicate values and constant elements. The multiplication of such quantities then poses a good example with relevance to operation on larger systems.

```
void multiply(REAL const A[3][3], REAL const B[3][3], REAL C[3][3])
{
  for(int i=0;i<3;i++)
  {
    for(int j=0;j<3;j++)
    {
      C[i][j] = A[i][0]*B[0][j] + A[i][1]*B[1][j] + A[i][2]*B[2][j];
    }
  }
}
```

Figure 7: A numerical 3x3 matrix multiply function

```
void main(void)
{
  // setup the problem
  REAL a[3][3];
  a[0][0] = 1.0; a[0][1] = 0.0; a[0][2] = 0.0;
  a[1][0] = 0.0; a[1][1] = 1.0; a[1][2] = 0.0;
  a[2][0] = 0.0; a[2][1] = 0.0; a[2][2] = 1.0;

  REAL b[3][3];
  b[0][0] = 10.0; b[0][1] = 0.5; b[0][2] =  0.0;
  b[1][0] =  0.5; b[1][1] = 5.1; b[1][2] =  0.7;
  b[2][0] =  0.0; b[2][1] = 0.7; b[2][2] = 10.0;

  REAL c[3][3];  // the intended output
  REAL q; // rotation angle

  // begin simulation loop

    // do simulation stuff

    // in this iteration q is 30 degrees
    q = 30.0 * PI / 180.0;

     // a is a direction cosine describing a simple rotation about the z
axis
    a[0][0] = a[1][1] = COS(q);
    a[0][1] = SIN(q);
    a[1][0] = -SIN(q);

    multiply(a,b,c);

    // inspect the output every iteration
    print(c);

  // end loop
}
```

Figure 8: A sample numerical mainline program.

A typical numerical three by three matrix multiplication function is shown in figure 7. Figure 8 illustrates a traditional mainline sample program where comments denote the overall structure of the program which is omitted for simplicity. This code runs and produces the following output in the terminal window (note that the *print()* function is not shown).

```
8.91025 2.98301 0.35
-4.56699 4.16673 0.606218
0 0.7 10
```

To capture the same numerical procedure symbolically, an alternate mainline program can be used which runs the simulation loop only once. Such a program is depicted in figure 9. It is important to point out that the symbolic procedure does not in general require custom coded mainline programs and simulations parameters can be completely automated with user input obtained in the same system description step shown previously in figure 1. Most importantly in this demonstration, no modifications have been made to the source code of the *multiply* function.

```
void main(void)
{
  REAL a[3][3];
  a[0][0] = 1.0; a[0][1] = 0.0; a[0][2] = 0.0;
  a[1][0] = 0.0; a[1][1] = 1.0; a[1][2] = 0.0;
  a[2][0] = 0.0; a[2][1] = 0.0; a[2][2] = 1.0;

  REAL b[3][3];
  b[0][0] = 10.0; b[0][1] = 0.5; b[0][2] =  0.0;
  b[1][0] =  0.5; b[1][1] = 5.1; b[1][2] =  0.7;
  b[2][0] =  0.0; b[2][1] = 0.7; b[2][2] = 10.0;

  REAL c[3][3];

  REAL q = create_runtime_const("q");

   // a is a direction cosine describing a simple rotation about the z
axis
  a[0][0] = a[1][1] = COS(q);
  a[0][1] = SIN(q);
  a[1][0] = -SIN(q);

  multiply(a,b,c);

  // identify the output (we only want the first column of c)
  for(int i=0;i<3;i++)
  {
    output(c[i][0]);
  }

  generate_code("my_file.c");
}
```

Figure 9: A mainline program for symbolic interpretation.

```
void preprocess(double const *parameters, double *constants)
{
}

void compute(double const *input, double const *constants, double
*output)
{
  double result[8];

  // the calculations
  res[0] = cos(input[0]);
  res[1] = sin(input[0]);
  res[2] = res[0] * 10;
  res[3] = res[1] * 0.5;
  res[4] = res[2] + res[3];
  res[5] = -res[1] * 10;
  res[6] = res[0] * 0.5;
  res[7] = res[5] + res[6]

  // align output
  output[0] = res[4];
  output[1] = res[7];
  output[3] = 0;
}
```

Figure 10: Symbolic code result file *my_file.c*.

Compiling and executing the symbolic mainline program results in a pair of functions being written to *my_file.c*. A listing of this file is shown in figure 10.

It is observed from the source code that there are no preprocessing calculations because no elements in the the solution were parameterized. And computation of the first column of the 3x3 matrix was reduced from 6 additions and 9 multiplications (and 2 trig functions) to 2 additions and 4 multiplications.

The generated code can then be compiled with another simple mainline program such as that shown in figure 11. This program produces the expected result in the terminal.

```
8.91025
-4.56699
0
```

A LARGE SCALE EXAMPLE AND ANALYSIS

At the deadline for this DRAFT paper we are still debugging the large scale implementation.

We will provide performance timings and comparisons for various size multibody systems auto coded from a recursive numerical algorithm.

A unique analysis will also be performed using the dependency graph to visualize the native parallelism present in the solution. Specifically, three plots will be presented. The first will be the number of calculations appearing at each level of depth in the dependency graph as ordered from the input. The second will instead use depth computed from the output. These two plots indicate the theoretical minimum number of cycles required for a parallel solution and the uneven nature of parallel resources which must be available at each step. These two plots represent two extremes and the third plot will

```
void main(void)
{
  double q = 30.0 * PI / 180.0;  // 30 degrees
  double c[3];

  preprocess(0,0);
  compute(&q,0,c);

  std::cout << c[0] << std::endl << c[1] << std::endl << c[2] << std::endl;
}
```

Figure 11: a mainline program using the auto-coded result.

demonstrate an attempt to smooth the peaks and fill the valleys without violating dependencies (resource optimization).

In practice a graph theoretic optimization approach could be applied (factoring in interprocessor communications) to produce parallelism custom tuned for an simulation platform or embedded target. Such a method would be expected to find and separate large blocks of independent calculations throughout the solution. Such features are the topic of future research.

FUTURE DEVELOPMENT

There are several points which require additional development to achieve increased efficiency in the symbolic interpreter and resulting auto-coded sources.

All operations shown in the code listing of figure 10 are binary operations and assignments. While the compiler is likely to optimize this, it is a relatively simple matter to inspect the dependency graph and to only allocate intermediate variables for quantities which appear more than once in the solution. Thereby permitting longer lines of code.

A smooth transition from symbolic calculation to numerical components (such as contact force calculation) and back is highly desirable. The symbolic calculations should be capable of integrating directly with library functions which are not enabled for symbolic processing.

It would be a simple matter for the *RealNumber* class to support conditionals in numerical code by carrying a default value. In this way pivoted inversion routines can be autocoded about known stable solutions. However, it is currently the preference to allow all such code to compile but to issue errors if such operations are encountered when symbolically interpreting code. That is to say that the optimizer can be used to generate generalized mass matrices, but should not be used for inversion of matrices which require pivoting.

Common conditionals expressions such as absolute value are given native symbolic representations. Support for a ternary style *if* statement is currently under consideration.

CONCLUSIONS

A software model for the automatic generation of symbolically optimized implementations from existing numerical code has been presented. A small numerical program representative of typical dynamics manipulations was successfully interpreted and the export of symbolically optimized code was directly demonstrated to both achieve the same result and reduce the number of operations. A large scale example with timings and parallel processing analysis is forthcoming (TBD for this paper).

ACKNOWLEDGMENTS

REFERENCES

[1] Rosenthal, D.E., and Sherman, M.A., "High performance multibody simulations via symbolic equation manipulation and Kane's method." Journal of the Astronautical Sciences 34, no. 3 (1986): 223-239.

[2] Schaechter, David B., David A. Levinson, and Thomas R. Kane. Autolev user's manual. Online Dynamics, 1988.

[3] Shi, Pengfei, and John McPhee. "DynaFlex Users Guide." Systems Design Engineering, University of Waterloo (2002).

[4] Johnson, Ralph, E. Gamma, R. Helm, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley 1, no. 1-2 (1995): 33-57.

[5] Butenhof, David R. Programming with POSIX threads. Addison-Wesley Professional, 1997.

[6] Plauger, Phillip James, Meng Lee, David Musser, and Alexander A. Stepanov. C++ Standard Template Library. Prentice Hall PTR, 2000.